Password Cracking: A Game of Wits

The following report has been gleaned from "A Tour of the Worm," an in-depth account of the November Internet infection. The author found the worm's crypt algorithm a frustrating, yet engaging, puzzle.

Donn Seeley

A password cracking algorithm seems like a slow and bulky item to put in a worm, but the worm makes this work by being persistent and efficient. The worm is aided by some unfortunate statistics about typical password choices.

> *For example, if the login name is "abc," then "abc," "cba," and "abcabc" are excellent candidates for passwords.*
>
> [F.T. Grammp and R. Morris]

The worm's password guessing is driven by a 4-state machine. The first state gathers password data, while the remaining states represent increasingly less likely sources of potential passwords. The central cracking routine is called cracksome( ), and it contains a switch on each of the four states.

The routine that implements the first state we named crack_0( ). This routine's job is to collect information about hosts and accounts. It is only run once; the information it gathers persists for the lifetime of the worm. Its implementation is straightforward: it reads the files */etc/hosts.equiv* and */.rhosts* for hosts to attack, then reads the password file looking for accounts. For each account, the worm saves the name, the encrypted password, the home directory and the user information fields. As a quick preliminary check, it looks for a *.forward* mail forwarding file in each user's home directory and saves any host name it finds in that file.

We unimaginatively called the function for the next state crack_1( ). .crack_1( ) looks for trivially broken passwords. These are passwords which can be guessed merely on the basis of information already contained in the password file. Grampp and Morris [2] report a survey of over 100 password files that found that between 8 and 30 percent of all passwords could be guessed using just the literal account name and a couple of variations. The worm tries a little harder than this: it checks the null password, the account name, the account name concatenated with itself, the first name (extracted from the user information field, with the first letter mapped to lower case), the last name, and the account name reversed. It runs through up to 50 accounts per call to cracksome( ), saving its place in the list of accounts and advancing to the next state when it runs out of accounts to try.

The next state is handled by crack_2( ). In this state the worm compares a list of favorite passwords, one password per call, with all of the encrypted passwords in the password file. The list contains 432 words, most of which are real English words or proper names; it seems likely that this list was generated by stealing password files and cracking them at leisure on the worm author's home machine. A global variable nextw is used to count the number of passwords tried, and it is this count (plus a loss in the population control game) that controls whether the worm exits at the end of the main loop—nextw must be greater than 10 before the worm can exit. Since the worm normally spends 2.5 minutes checking for clients over the course of the main loop and calls cracksome( ) twice in that period, it appears that the worm must make a minimum of 7 passes through the main loop, taking more than 15 minutes.[1] It will take at least nine hours for the

[1] For those mindful of details: the first call to cracksome( ) is consumed reading system files. The worm must spend at least one call to cracksome( ) in the second state attacking trivial passwords. This accounts for at least one pass through the main loop. In the third state, cracksome( ) tests one password from its list of favorites on each call; the worm will exit if it lost a roll of the dice and more than 10 words have been checked, so this accounts for at least six loops, two words on each loop for five loops to reach 10 words, then another loop to pass that number. Altogether this amounts to a minimum of 7 loops. If all 7 loops took the maximum amount of time waiting for clients, this would require a minimum of 17.5 minutes, but the two-minute check can exit early if a client connects and the server loses the challenge, hence 15.5 minutes of waiting time plus runtime overhead is the minimum lifetime. In this period, a worm will attack at least 8 hosts through the host infection routines, and will try about 18 passwords for each account, attacking more hosts if accounts are cracked.

worm to scan its built-in password list and proceed to the next state.

The last state is handled by crack_3( ). It opens the UNIX® online dictionary */usr/dict/words* and goes through it one word at a time. If a word is capitalized, the worm tries a lower-case version as well. This search can essentially go on forever: it would take something like four weeks for the worm to finish a typical dictionary like ours.

When the worm selects a potential password, it passes it to a routine we called try_password( ). This function calls the worm's special version of the UNIX password encryption function crypt( ) and compares the result with the target account's actual encrypted password. If they are equal, or if the password and guess are the null string (no password), the worm saves the cleartext password and proceeds to attack the hosts that are connected to this account. A routine we called try_forward_and_rhosts( ) reads the user's *.forward* and *.rhosts* files, calling the previously described hu1( ) function for each remote account it finds.

FASTER PASSWORD ENCRYPTION

The use of encrypted passwords appears reasonably secure in the absence of serious attention of experts in the field.

[R. Morris and K. Thompson]

Unfortunately, some experts in the field have been giving serious attention to fast implementations of the UNIX password encryption algorithm. UNIX password authentication works without putting any readable version of the password onto the system, and indeed works without protecting the encrypted password against reading by users on the system. When a user types a password in the clear, the system encrypts it using the standard crypt( ) library routine, then compares it against a saved copy of the encrypted password. The encryption algorithm is meant to be basically impossible to invert, preventing the retrieval of passwords by examining only the encrypted text, and it is meant to be expensive to run, so that testing guesses will take a long time. The UNIX password encryption algorithm is based on the Federal Data Encryption Standard (DES). Currently no one knows how to invert this algorithm in a reasonable amount of time, and while fast DES encoding chips are available, the UNIX version of the algorithm is slightly perturbed so that it is impossible to use a standard DES chip to implement it.

Two problems have been mitigating against the UNIX implementation of DES. Computers are continually increasing in speed—current machines are typically several times faster than the machines that were available when the current password scheme was invented. At the same time, methods have been discovered to make software DES run faster. UNIX passwords are now far more susceptible to persistent guessing, particularly if the encrypted passwords are already known. The worm's version of the UNIX crypt( ) routine ran more than nine times faster than the standard version when we tested it on our VAX 8600. While the standard crypt( ) takes 54 seconds to encrypt 271 passwords on our 8600 (the number of passwords actually contained in our password file), the worm's crypt( ) takes less than six seconds.

The worm's crypt( ) algorithm appears to be a compromise between time and space: the time needed to encrypt one password guess versus the substantial extra table space needed to squeeze performance out of the algorithm. Curiously, one performance improvement actually saves a little space. The traditional UNIX algorithm stores each bit of the password in a byte, while the worm's algorithm packs the bits into two 32-bit words. This permits the worm's algorithm to use bit-field and shift operations on the password data, which are immensely faster. Other speedups include unrolling loops, combining tables, precomputing shifts and masks, and eliminating redundant initial and final permutations when performing the 25 applications of modified DES that the password encryption algorithm uses. The biggest performance improvement comes as a result of combining permutations: the worm uses expanded arrays which are indexed by groups of bits rather than the single bits used by the standard algorithm. Matt Bishop's fast version of crypt( ) [1] does all of these things and also precomputes even more functions, yielding twice the performance of the worm's algorithm but requiring nearly 200 KB of initialized data as opposed to the 6 KB used by the worm and the less than 2 KB used by the normal crypt( ).

How can system administrators defend against fast implementations of crypt( )? One suggestion that has been introduced for foiling the bad guys is the idea of shadow password files. In this scheme, the encrypted passwords are hidden rather than public, forcing a cracker to either break a privileged account or use the host's CPU and (slow) encryption algorithm to attack, with the added danger that password test requests could be logged and password cracking discovered. The disadvantage of shadow password files is that if the bad guys somehow get around the protections for the file that contains the actual passwords, all of the passwords must be considered cracked and will need to be replaced.

Another suggestion has been to replace the UNIX DES implementation with the fastest available implementation, but run it 1000 times or more instead of the 25 times used in the UNIX crypt( ) code. Unless the repeat count is somehow pegged to the fastest available CPU speed, this approach merely postpones the day of reckoning until the cracker finds a faster machine. It's interesting to note that Morris and Thompson measured the time to compute the old M-209 (non-DES) password encryption algorithm used in early versions of UNIX on the PDP-11/70 and found that a good implementation took only 1.25 milliseconds per encryption, which they

UNIX is a registered trademark of AT&T Bell Laboratories.

VAX is a trademark of Digital Equipment Corporation.

deemed insufficient; currently the VAX 8600 using Matt Bishop's DES-based algorithm needs 11.5 milliseconds per encryption, and machines 10 times faster than the VAX 8600 at a cheaper price will be available soon (if they aren't already!).

OPINIONS

> *The act of breaking into a computer system has to have the same social stigma as breaking into a neighbor's house. It should not matter that the neighbor's door is unlocked.*
>
> [K. Thompson]

> [*Creators of viruses are*] *stealing a car for the purpose of joyriding.* [R. Morris, in 1983 Capitol Hill testimony, cited in the *New York Times*, 11/11/88]

I do not propose to offer definitive statements on the morality of the worm's author, the ethics of publishing security information or the security needs of the UNIX computing community, since people better (and less) qualified than I are still copiously flaming on these topics in the various network newsgroups and mailing lists. For the sake of the mythical ordinary system administrator who might have been confused by all the information and misinformation, I will try to answer a few of the most relevant questions in a narrow but useful way.

Did the worm cause damage? The worm did not destroy files, intercept private mail, reveal passwords, corrupt databases or plant trojan horses. It did compete for CPU with, and eventually overwhelm, ordinary user processes. It used up limited system resources such as the open file table and the process text table, causing user processes to fail for lack of same. It caused some machines to crash by operating them close to the limits of their capacity, exercising bugs that do not appear under normal loads. It forced administrators to perform one or more reboots to clear worms from the system, terminating user sessions and long-running jobs. It forced administrators to shut down network gateways, including gateways between important nation-wide research networks in an effort to isolate the worm. This action led to delays of up to several days in the exchange of electronic mail, causing some projects to miss deadlines and others to lose valuable research time.

It made systems staff across the country drop their ongoing hacks and work 24-hour days trying to corner and kill worms. It caused members of management in at least one institution to become so frightened that they scrubbed all the disks at their facility that were online at the time of the infection, and limited reloading of files to data that was verifiably unmodified by a foreign agent. It caused bandwidth through gateways that were still running after the infection started to become substantially degraded—the gateways were using much of their capacity just shipping the worm from one network to another. It penetrated user accounts and caused it to appear that a given user was disturbing a system when in fact they were not responsible. It's true that the worm could have been far more harmful than it actually turned out to be: in the last few weeks, several security bugs have come to light which the worm could have used to thoroughly destroy a system. Perhaps we should be grateful that we escaped incredibly awful consequences, and perhaps we should also be grateful that we have learned so much about the weaknesses in our system's defenses, but I think we should share our gratefulness with someone other than the worm's author.

Was the worm malicious? Some people have suggested that the worm was an innocent experiment that got out of hand, and that it was never intended to spread so fast or so widely. We can find evidence in the worm to support and to contradict this hypothesis. There are a number of bugs in the worm that appear to be the result of hasty or careless programming. For example, in the worm's if_init( ) routine, there is a call to the block zero function bzero( ) that incorrectly uses the block itself rather than the block's address as an argument. It's also possible that a bug was responsible for

the ineffectiveness of the population control measures used by the worm. This could be seen as evidence that a development version of the worm "got loose" accidentally, and perhaps the author originally intended to test the final version under controlled conditions, in an environment from which it would not escape.

On the other hand, there is considerable evidence that the worm was designed to reproduce quickly and spread itself over great distances. It can be argued that the population control hacks in the worm are anemic by design: they are a compromise between spreading the worm as quickly as possible and raising the load enough to be detected and defeated. A worm will exist for a substantial amount of time and will perform a substantial amount of work even if it loses the roll of the (imaginary) dice; moreover, one-in-seven worms become immortal and cannot be killed by dice rolls.

There is ample evidence that the worm was designed to hamper efforts to stop it even after it was identified and captured. It certainly succeeded in this, since it took almost a day before the last mode of infection (the finger server) was identified, analyzed and reported widely; the worm was very successful in propagating itself during this time even on systems which had fixed the sendmail debug problem and had turned off rexec. Finally, there is evidence that the worm's author deliberately introduced the worm to a foreign site that was left open and welcome to casual outside users, rather ungraciously abusing this hospitality. He apparently further abused this trust by deleting a log file that might have revealed information that could link his home site with the infection. I think the innocence lies in the research community rather than with the worm's author.

Will publication of worm details further harm security? In a sense, the worm itself has solved that problem: it has published itself by sending copies to hundreds or thousands of machines around the world. Of course, a bad guy who wants to use the worm's tricks would have to go through the same effort that we went through in order to understand the program, but then it only took us a week to completely decompile the program. Therefore, while it takes fortitude to hack the worm, it clearly is not greatly difficult for a decent programmer. One of the worm's most effective tricks was advertised when it entered—the bulk of the sendmail hack is visible in the log file, and a few minutes work with the sources will reveal the rest of the trick. The worm's fast password algorithm could be useful to the bad guys, but at least two other faster implementations have been available for a year or more, so it is not very secret, or even very original. Finally, the details of the worm have been well enough sketched out on various newsgroups and mailing lists that the principal hacks are common knowledge. I think it is more important that we understand what happened, so that we can make it less likely to happen again, rather than spend time in a futile effort to cover up the issue from everyone but the bad guys. Fixes for both source and binary distributions are widely available, and anyone who runs a system with these vulnerabilities needs to look into these fixes immediately, if they have not done so already.

CONCLUSION

It has raised the public awareness to a considerable degree.

[R. Morris, *New York Times*, 11/5/88]

This quote is one of the understatements of the year. The worm story was on the front page of the *New York Times* and other newspapers for days. It was the subject of television and radio features. Even the *Bloom County* comic strip poked fun at it.

Our community has never before been in the limelight in this way, and judging by the response, it has scared us. I will not offer any fancy platitudes about how the experience is going to change us, but I will say that I think these issues have been ignored for much longer than was safe, and I feel that a better understanding of the crisis just past will help us cope better with the next one. Let's hope we are as lucky the next time.

REFERENCES

1. Bishop, M. A fast version of the DES and a password encryption algorithm. Universities Space Research Institute for Advanced Computer Science. NASA Ames Research Center, Moffett Field, CA.
2. Grampp, F.T., and Morris, R. UNIX operating system security. *AT&T Bell Laboratories Tech. J. 63*, 8, Part 2, (Oct. 1984), 1649.
3. Morris, R., and Thompson, K. Password Security: A Case History, dated April 3, 1978, in the UNIX Programmer's Manual; in the Supplementary Documents or the System Manager's Manual, (depending upon source and date of manuals).
4. Seeley, D. A tour of the worm. Proceedings of the Winter 1989 Usenix Conference. San Diego, CA, p. 287.
5. Thompson, K. Reflections on trusting trust, 1983 ACM Turing Award Lecture. *Commun. ACM 27*, 8 (Aug. 1984), 761.

CR Categories and Subject Descriptors: C.2.0 [**Computer Communication Networks**]: General—*security and protection*; D.4.6 [**Operating Systems**]: Cryptographic Controls; K.4.2 [**Computers and Society**]: Social Issues—*abuse and crime involving computers*

General Terms: Security

Additional Key Words and Phrases: UNIX, viruses, worms

ABOUT THE AUTHOR:

DONN SEELEY is a member of the systems staff for the Department of Computer Science at the University of Utah in Salt Lake City. He has contributed to the U.S. Berkeley Software Distribution of the UNIX operating system, and works on UNIX compilers. Author's Present Address: Department of Computer Science, 3190 Merrill Engineering Building, University of Utah, Salt Lake City, UT 84112. donn@cs.utah.edu.